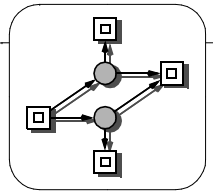
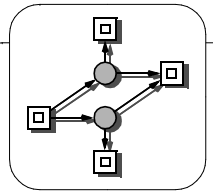


SOFTWARE MODELLING WITH PETRI NETS

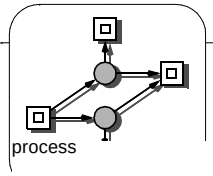


STEP-WISE MODELLING

- according system structure
  - -> *each sequential process separately*
  - -> *composition of processes*

    system of interacting concurrent processes (ICP)
- according process structure
  - -> *control structure model (csm)*
    as place/transition net
  - -> *control flow model (cfm)*
    as place/transition net or
    as coloured net

    cfm = csm + control variables
  - -> *data flow model*
- abstractions
  - -> *data,*
    extend depends on model purpose
  - -> *time*



CSM REFERENCE LANGUAGE

process ::= process_id@ ":" **process**
statement_sequence
#process process_id@ .

statement_sequence ::= statement_sequence ";" statement
| statement .

statement ::= if_statement
| case_statement
| loop_statement
| jump_statement
| send_statement
| receive_statement
| wait_statement
| simple_statement@ .

if_statement ::= **if** if_expression@
then statement_sequence
[**else** statement_sequence]
#if .

case_statement ::= **case** case_expression@ **of**
case_label@ ":" statement_sequence
{ "**|**" case_label@ ":" statement_sequence }*
[**default** ":" statement_sequence]
#case .

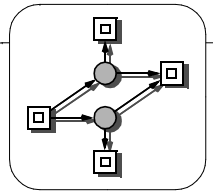
loop_statement ::= [loop_ label@ **"**:"]
loop [loop_expression@]
statement_sequence
#loop [loop_ label@] .

jump_statement ::= **next** loop_label@
| **exit** loop_label@
| **stop** .

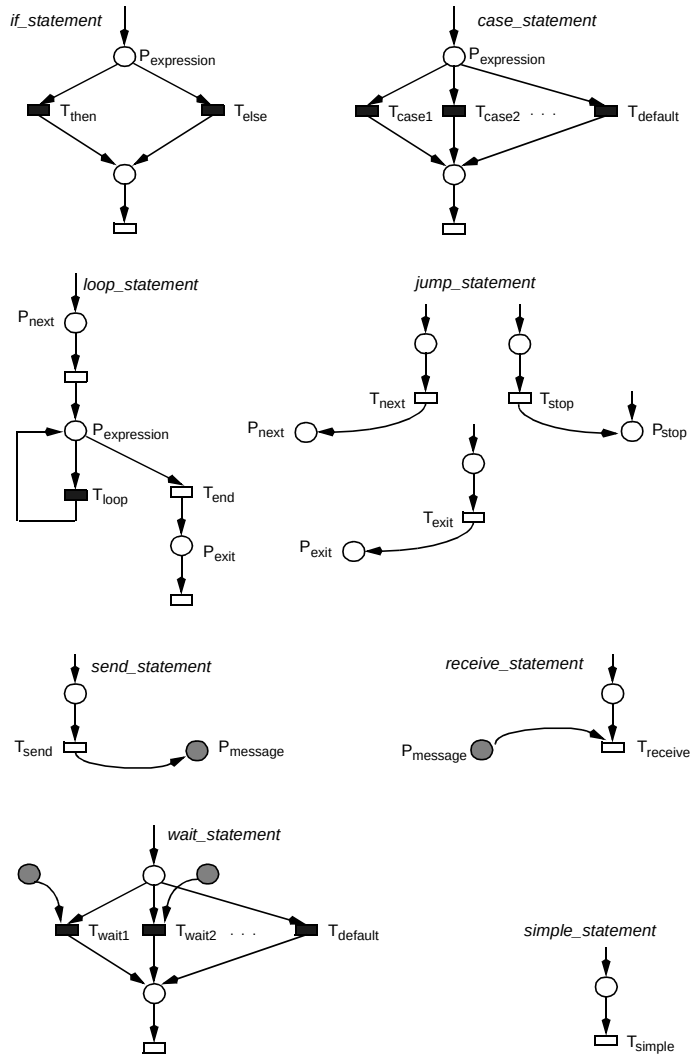
send_statement ::= **send** message@ [**to** process_id@] .

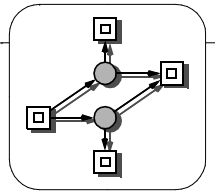
receive_statement ::= **receive** message@ [**from** process_id@] .

wait_statement ::= **wait**
message@ [**from** process-id@] ":" statement_sequence
{ "|" message@ [**from** process-id@] ":" statement_sequence }*
#wait .



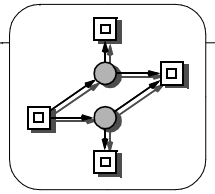
CSM, PETRI NET COMPONENTS





SEQUENTIAL PROCESS, CONTROL STRUCTURE MODEL (WITHOUT DATA DEPENDENCIES)

- pure
- ordinary
  - homogenous
- conservative
  - structurally bounded
- marked with exactly one token
  - safe (1-bounded)
- CPI
- all static conflicts are dynamically realizable
- SM,
  SCSM (if only structured goto's)



CFM REFERENCE LANGUAGE (IN ADDITION TO CSM GRAMMAR)

case_expression@ :: = Bool_expression
| ordinal_expression@.

loop_expression@ :: = Bool_expression
| ordinal_expression@ .

Bool_expression :: = Bool_operand
| **not** Bool_expression
| Bool_expression **or** Bool_operand
| Bool_expression **and** Bool_operand
| Bool_expression "=" Bool_operand
| Bool_expression "/=" Bool_operand .

Bool_operand :: = Bool_denotation
| Bool_variable
| "(" Bool_expression ")" .

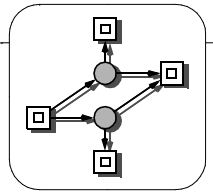
Bool_denotation :: = **true**
| **false** .

Bool_variable :: = identifier@ { of type Boolean } .

statement :: = Bool_declaration
| Bool_assignation .

Bool_declaration :: = **Bool** Bool_variable ":=" Bool_denotation .

Bool_assignation :: = Bool_variable ":=" Bool_expression .

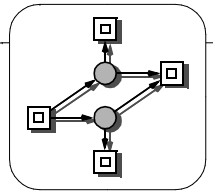


REMARKS TO CFM GRAMMAR

- each declaration of a Boolean variable encloses its initialization
- grammar rules defining the same metanotion are additive
- for each variable, the usual data flow restrictions of static semantics apply to the order of declarative and applied occurrence
- Boolean expressions can be of any complexity

DECLARATION OF A BOOLEAN VARIABLE A

- two places A and /A are generated
- initial marking according to the given initialization of the variable



NO ZERO TEST IN P/T NETS!

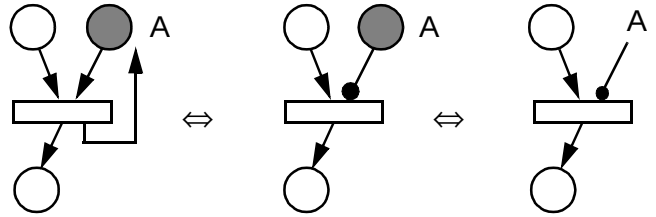
- wayout for finite, discrete data types

  each variable is modelled by as many places as there are values in the type,

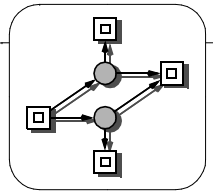
  e. g. for a Boolean variable P (initializid with true)

  P / P

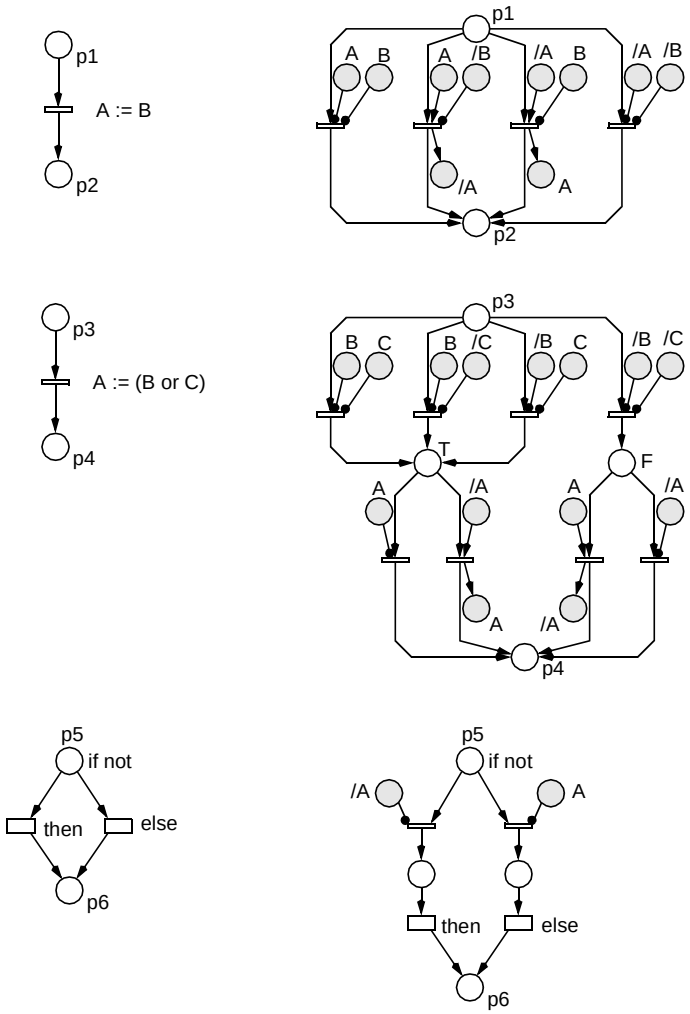
- notation agreement for test edge

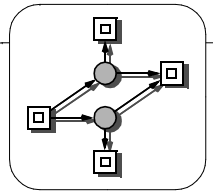


A - connector (fusion node)



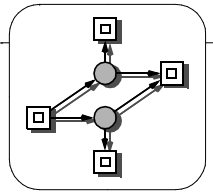
P/T NET COMPONENTS FOR BOOLEAN OPERATIONS, EXAMPLES



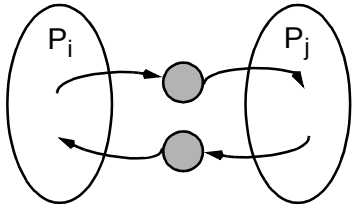


SEQUENTIAL PROCESS, CONTROL FLOW MODEL (WITH DATA DEPENDENCIES)

- not pure
- ordinary
  - ↷ homogenous
- conservative
  - ↷ structurally bounded
- not marked with exactly one token
  - one process counter
  - one token for each pair of Boolean places
- CPI (1-P-invariants)
  - ↷ safe (1-bounded)
- no dynamic conflicts
  - ↷ max outdegree of a reachability graph node = 1
- not ES

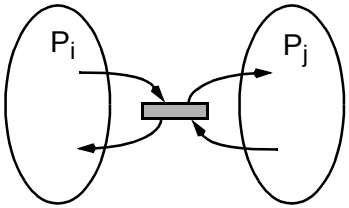


BASIC PRINCIPLES OF PROCESS CONNECTION

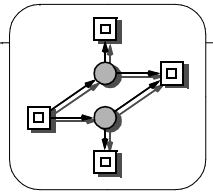


ICP

CSM - COMMUNICATING STATE MACHINES



ICP - INTERACTING CONCURRENT PROCESSES



PROCESS COMMUNICATION LANGUAGE CONSTRUCTS, A CLASSIFICATION

PROCESS COMMUNICATION

ADDRESSING

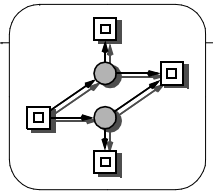
- **direct** *(one-to-one communication: sender and receiver know each other)*
- **semi-direct-by-sender** *(many-to-one communication: only the sender knows the receiver, not vice versa)*
- **semi-direct-by-receiver** *(one-to-many communication: only the receiver knows the sender, not vice versa)*
- **indirect** *(many-to-many communication: via common global objects like channels, mail boxes, monitors)*

SYNCHRONIZATION (of sender)

- **asynchronous** *(no-wait-send, the general case requires infinite buffer)*
- **semi-asynchronous** *(delay, if finite buffer full)*
- **(simple) synchronous** *(delay until message has been received)*
- **remote invocation** *(delay until a response has been given)*
- **hand shaking** *(delay until message has been exchanged, no buffering, direct transfer)*

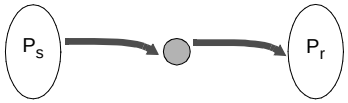
WAITING (of receiver)

- **deterministic** *(the choice of the message to receive occurs independently from the progress of neighbouring processes)*
- **non-deterministic** *(receiving is influenced by the available messages provided by neighbouring processes)*

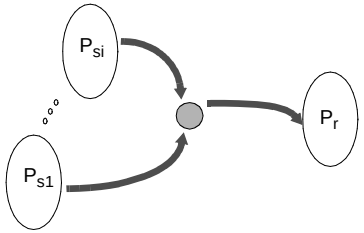


PROCESS ADDRESSING, PETRI NET COMPONENTS

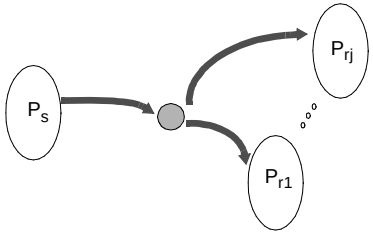
direct
(only
static channel conflicts)



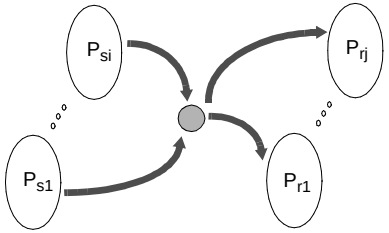
semi-direct-by-sender
(only
static channel conflicts)



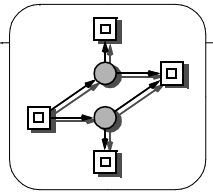
semi-direct-by-receiver
(dynamic channel conflicts
possible)



indirect
(dynamic channel conflicts
possible)

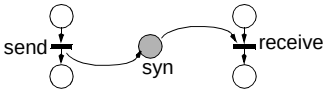


The pre-process P_s can send from different control points, and the post-process P_r can receive from different control points.

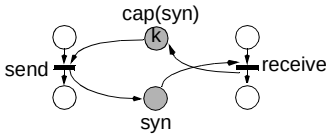


PROCESS SYNCHRONIZATION, PETRI NET COMPONENTS

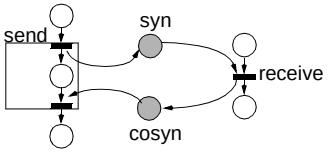
asynchronous
(possibly unbounded)



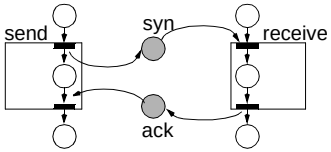
semi-asynchronous
(k-bounded,
locally conservative)



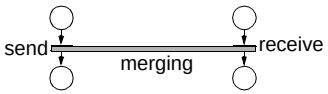
synchronous
(safe,
globally conservative)

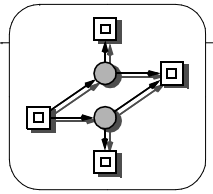


remote invocation
(safe,
globally conservative)



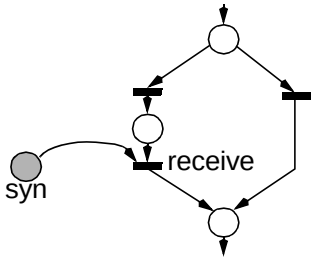
hand-shaking
(safe,
locally conservative)



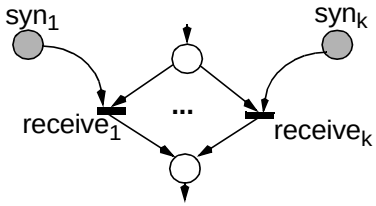


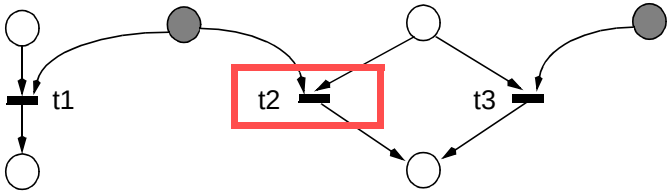
PROCESS WAITING, PETRI NET COMPONENTS

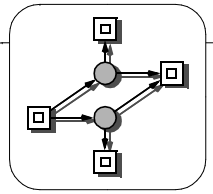
deterministic
(confusion impossible)
pure control flow conflicts)



non-deterministic
(confusion possible)

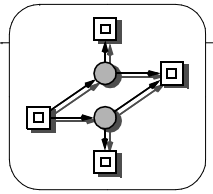






MODEL OF INTERACTING CONCURRENT PROCESSES

- ordinary
  - homogeneous
- safe (1-bounded), if only
  synchronous or
  remote invocation or
  rendezvous communication
- bounded, if only
  semi-asynchronous communication

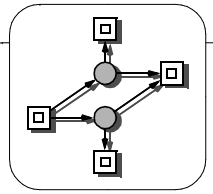


SIMPLIFIED VIEW[1] ON THE INFLUENCE OF COMMUNICATION PATTERNS ON NET STRUCTURE CLASS

waiting \ addressing	direct / semi-direct-by-sender	indirect / semi-direct-by-receiver
determininistic	EFC	ES
non-deterministic	ES	ISP

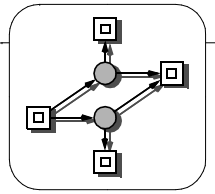
1. provided, pre- and postprocesses do not access the same communication object from different control points

known to be time-independently live [Starke 90], i.e. a live net remains live under any constant delay timing (duration net).

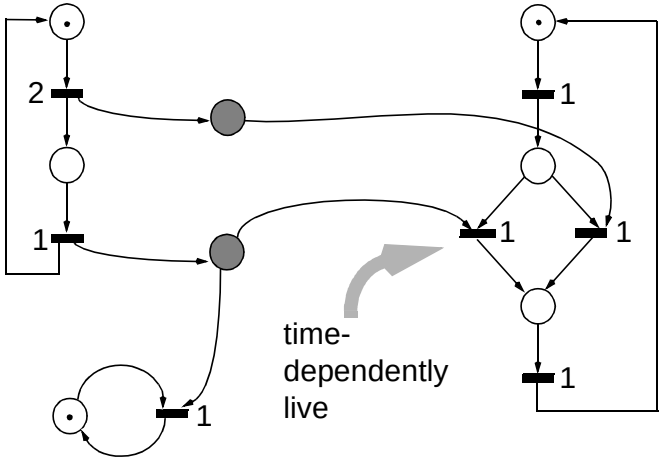


THE INFLUENCE OF COMMUNICATION PATTERNS ON CONFLICT STRUCTURES

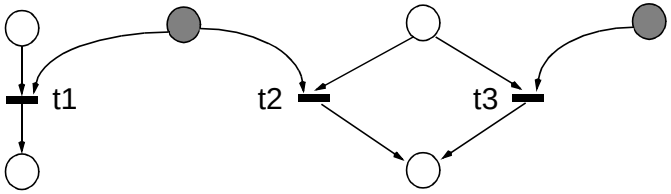
waiting \ addressing	direct / semi-direct-by-sender	indirect / semi-direct-by-receive
deterministic	no dynamic channel conflicts	channel & control flow conflicts appear only separately
non-deterministic		confusing combination of channel & control flow conflicts possible

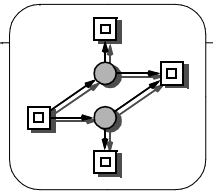


EXAMPLE OF A TIME-DEPENDENTLY LIVE (DURATION) CSN



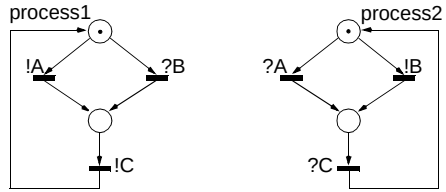
CONFUSING COMBINATION OF CHANNEL AND CONTROL FLOW CONFLICT



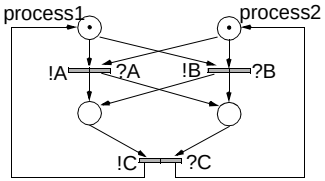


A SIMPLE PROTOCOL IN THREE VARIANTS

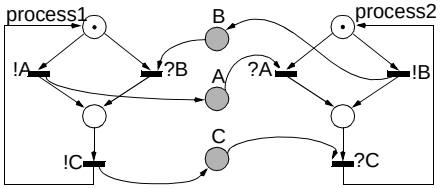
(1) given original process patterns



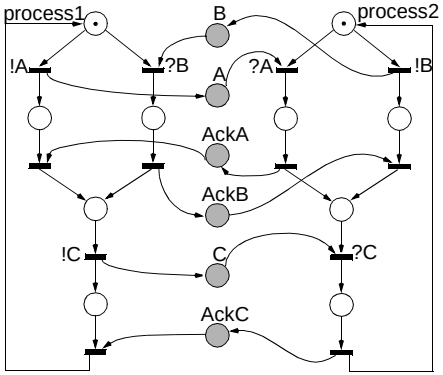
(1a) **rendezvous:** EFC, safe, live



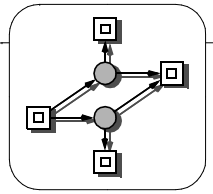
(1b) **asynchronous:** ES, unbounded, live



(1c) **remote invocation:** ES, safe, not live

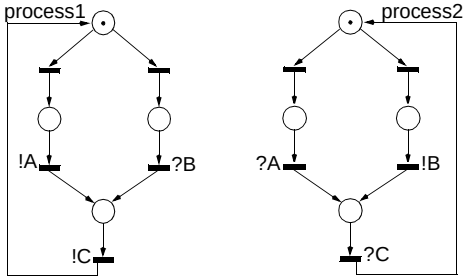


[Diaz 1986,LNCS 255]

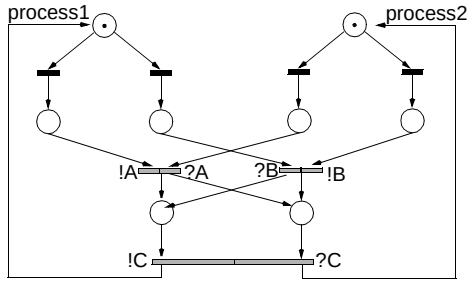


A MODIFIED SIMPLE PROTOCOL IN THREE VARIANTS

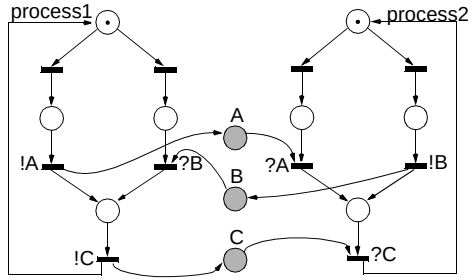
(2) given modified process patterns



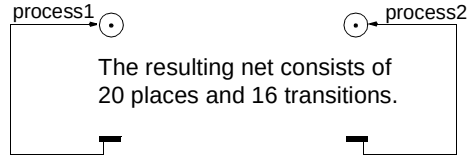
(2a) **rendezvous:** (E)FC, safe, not live

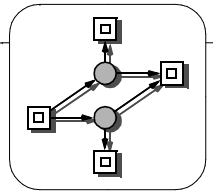


(2b) **asynchronous:** (E)FC, unbounded, not live



(2c) **remote invocation:** (E)FC, safe, not live





PETRI NET GENERATOR

- preconditions
  - -> *dedicated language constructs for all process interactions*
  - -> *(quasi-) static amount of processe*
  - -> *no run-time dependencies (like recursion, dynamic loops)*
- generated granularity
  - -> *coarse-grained control structure (communication skeleton, purely sequential program parts -> transitions*
  - -> *fine-grained control structure (statement structure, each statement -> transition)*
- support of cross referencing between program and net structure: node names with source text line numbers
- automatic layout of a sequential process' structure
- program complexity measure: Number of Acyclic Paths - NAP (number of structurally possible paths)